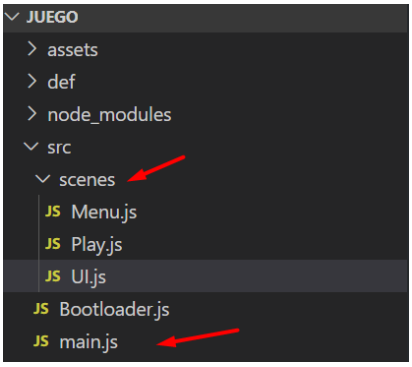
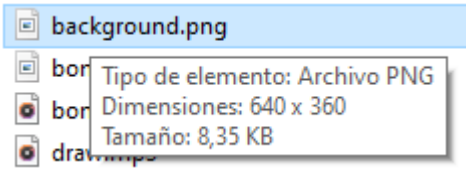
Juego con Phaser 3. “El karateka del pan con manteca”.

Vamos a tener la siguiente estructura de directorios o carpetas para nuestro juego:



El archivo “main” que guarda la configuración de nuestro juego. En el parámetro de anchura y altura ponemos las dimensiones de la imagen de fondo:



Observa en la imagen de la derecha que el tamaño de la imagen de fondo se traslada a las dimensiones del canvas de nuestro juego en la constante “config” Para las físicas del juego, establecimos las de “arcade” con una gravedad de 2000.

```
scale: {
    parent: "phaser_container",
    width: 640,
    height: 360,
    mode: Phaser.Scale.FIT,
    autoCenter: Phaser.Scale.CENTER_BOTH
},
pixelArt: true,
physics: {
    default: "arcade",
    "arcade": {
        gravity: {
            y: 2000
        }
    }
},
```

En la carpeta “scenes” vamos a crear las siguientes escenas:

- UI.js
- Play.js
- Menu.js

Dentro del archivo “**UI.js**”, vamos a crear una clase que extiende de la clase “Scene” de Phaser:

En ella crearemos el constructor, el método “init” y el método “create”:

En el constructor crearemos en la clase superior “super” la “key” a través de un json, y la llamaremos “UI”.

```
class UI extends Phaser.Scene{

}
```

No podemos olvidar exportar la clase con “**export default UI;**”

Esta estructura la vamos a copiar en los otros dos archivos creados: “Menu.js” y “Player.js”

```
class UI extends Phaser.Scene{
    constructor(){
        super({key: 'UI'});
    }

    init(){
        console.log('Se inicio la escena UI');
    }

    create(){

    }
}

export default UI;
```

```
class Menu extends Phaser.Scene{
    constructor(){
        super({key: 'Menu'});
    }

    init(){
        console.log('Se inicio la escena Menu');
    }

    create(){

    }
}

export default Menu;
```

```
class Play extends Phaser.Scene{
    constructor(){
        super({key: 'Play'});
    }

    init(){
        console.log('Se inicio la escena Play');
    }

    create(){

    }

    update(){

    }
}

export default Play;
```

A la escena "**Play**" le hemos añadido el **método "update"**. En el archivo "Main" debemos importar los archivos e incluir las escenas en el Array "scene" de la configuración (const config).

```
import Bootloader from './Bootloader.js';
import UI from './scenes/UI.js';
import Play from './scenes/Play.js';
import Menu from './scenes/Menu.js';
```

```
scene: [
    Bootloader,
    Play,
    UI,
    Menu
]
```

Cargamos en la escena "Bootloader.js" en "preload", las imágenes del juego con un Array:

```
preload() {
    this.load.path = './assets/';

    this.load.image([
        'background',
        'floor',
        'wall',
        'bomb',
        'toast_item',
        'life',
        'logo'
    ]);
```

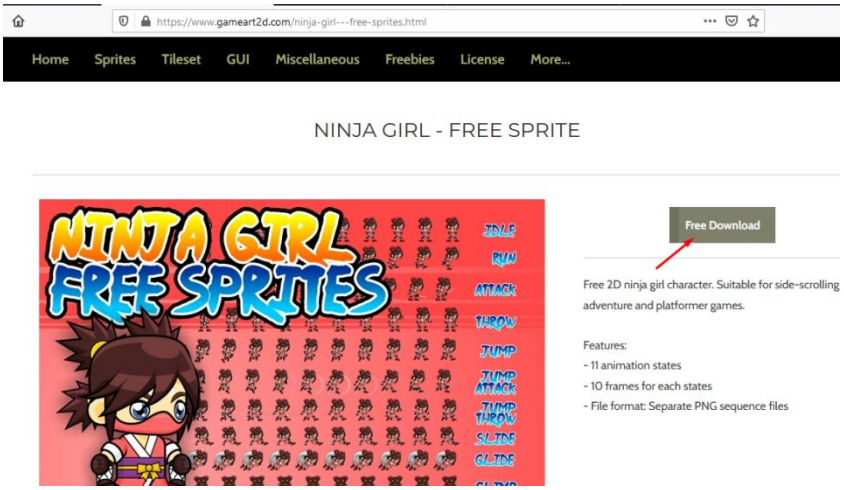
Cargamos los audios también:

```
this.load.audio('bongo', 'bongojam_f.mp3');
this.load.audio('pop', 'pop.mp3');
this.load.audio('draw', 'draw.mp3');
```

A continuación cargamos la fuente (usaremos una fuente con un atlas de fuentes; archivo "json"):

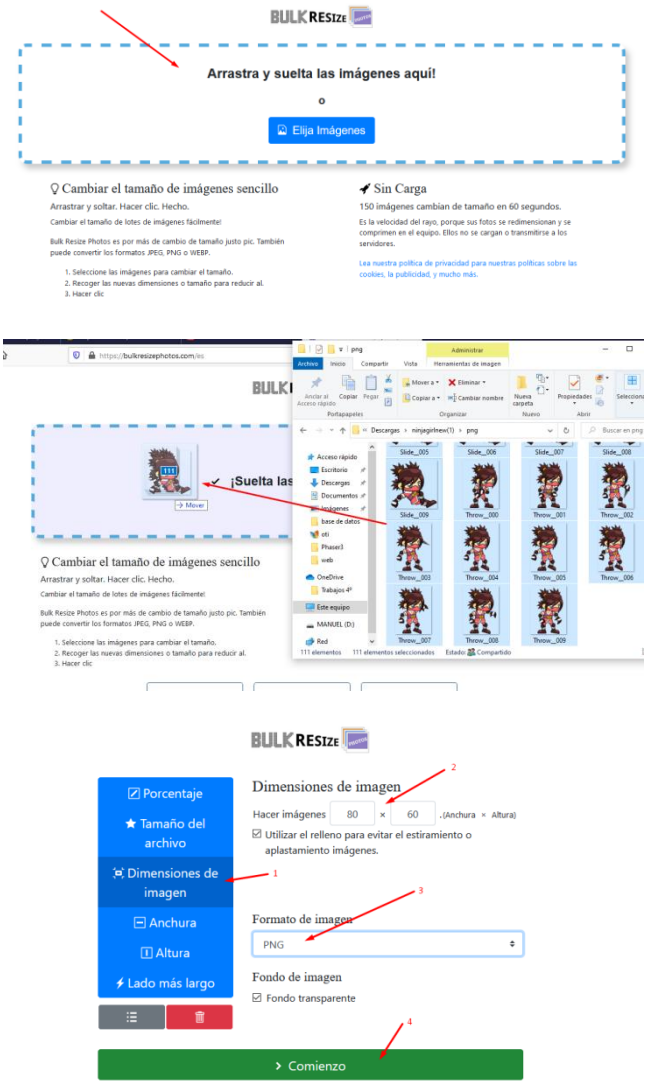
```
this.load.image('font', 'font/font.png');
this.load.json('fontData', 'font/font.json');
```

Para el personaje elijo un sprite gratuito y lo descargo:

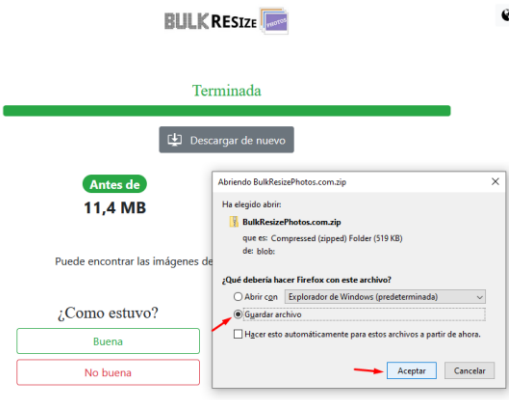


Descomprimo el archivo con las imágenes.

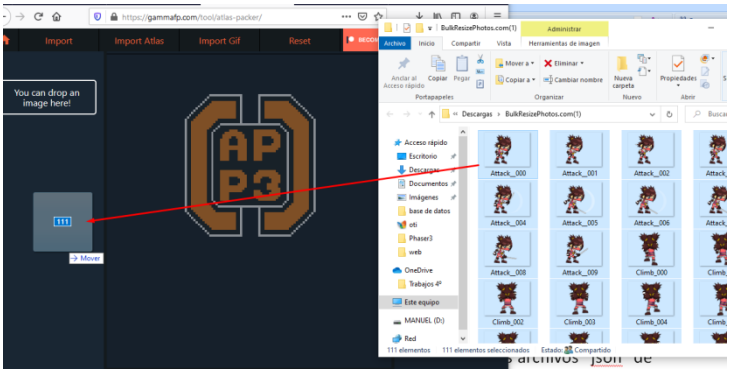
Usando la web https://bulkresizephotos.com/es redimensiono las fotos porque son muy grandes:



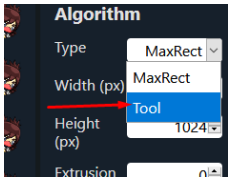
Me descargo la carpeta comprimida con las nuevas imágenes redimensionadas y la descomprimo.



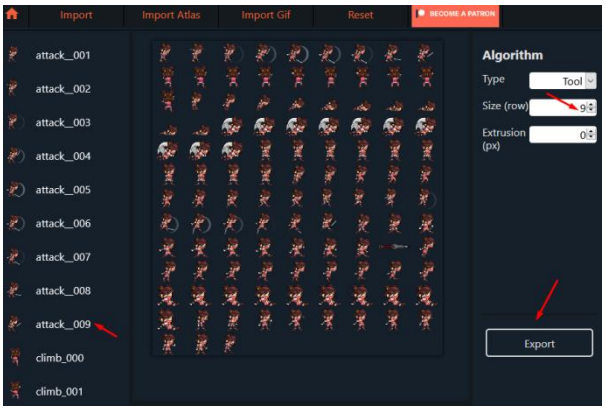
Usando la herramienta web https://gammafp.com/tool/atlas-packer/ Importo todas las imágenes:

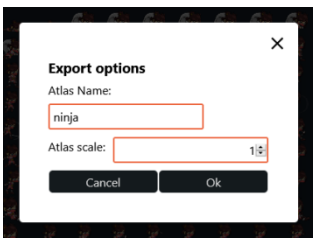


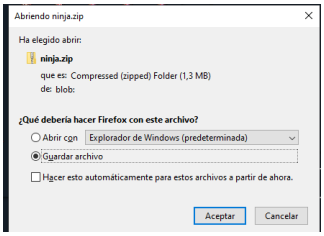
Elige la opción “Tool”:



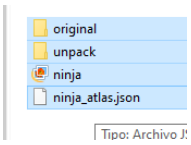
Elegí 9 columnas por ser 9 los frames de la animación. Exporto el atlas del Sprite:



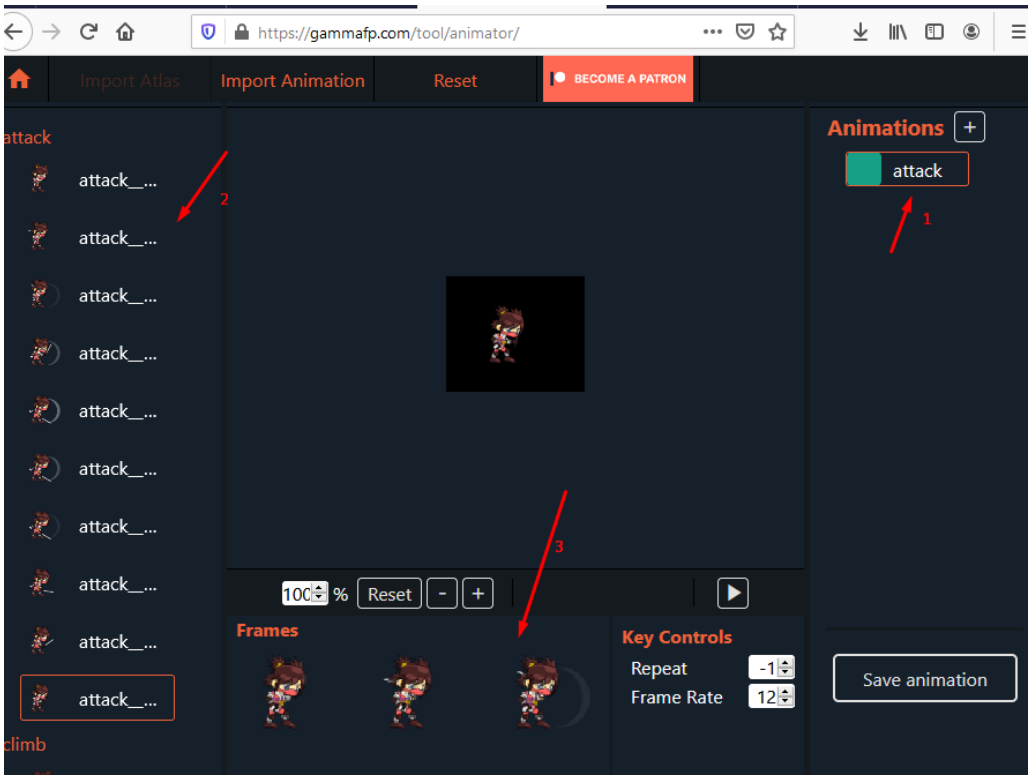




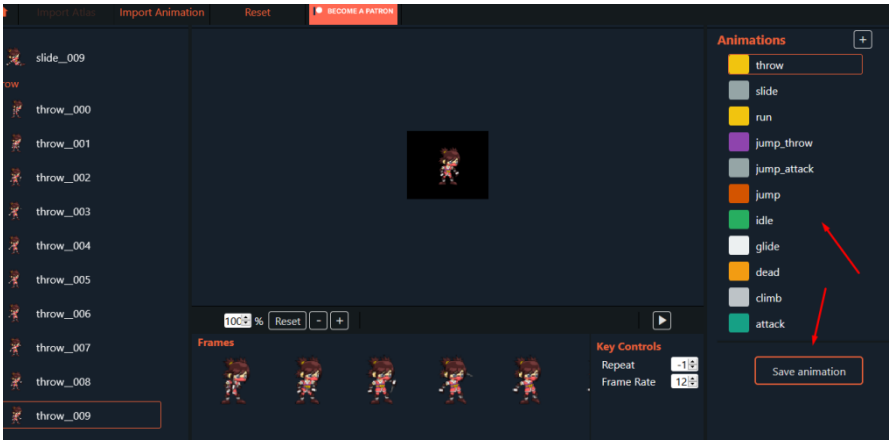
Al descomprimir la carpeta guardo los archivos en la carpeta “assets” del proyecto en una carpeta llamada “ninja”:

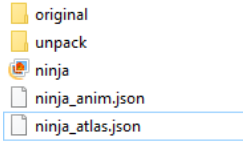


Y con la herramienta https://gammafp.com/tool/animator/ creo los archivos “json” de las animaciones eligiendo los dos archivos anteriores “ninja.png” y “ninja_atlas.json”:



Creo todas las animaciones pulsando el botón “+” a la derecha de “Animations”. Las animaciones deben llamarse como el prefijo (por ejemplo la primera “attack”) y vas añadiendo los frames de la izquierda (2). Cuando tengas todas las animaciones le das al botón “Save animation”.





En la carpeta “ninja” tienes que tener los archivos “ninja.png”, “ninja_atlas.json” y “ninja_anim.json”.

De forma que tenemos los “json” de los atlas y de las animaciones. Las incluyo en el “preload”:

```
this.load.atlas('ninja', 'ninja/ninja.png', 'ninja/ninja_atlas.json');
this.load.animation('ninjaAnim', 'ninja/ninja_anim.json');
```

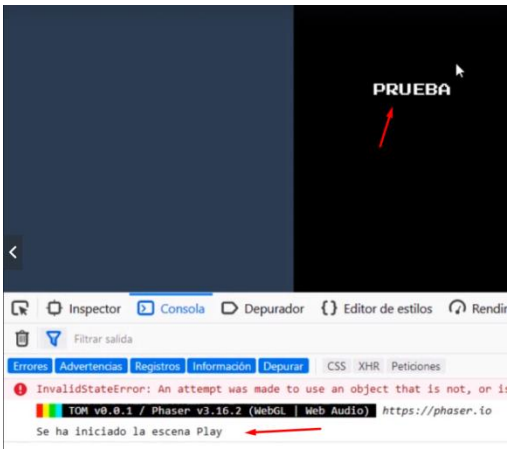
Para poder utilizar la fuente disponible y que se facilita en la carpeta del proyecto debemos incluirla en el método “preload” cuando finalice de cargar, usando para ello el evento “complete” y ejecutando la siguiente función anónima:

```
this.load.on('complete', () => {
    const fontData = this.cache.json.get('fontData');
    this.cache.bitmapFont.add('pixelFont', Phaser.GameObjects.RetroFont.Parse(this, fontData))

    this.scene.start('Play');
});
```

“fontData” es como hemos llamado al archivo “json” de la configuración de la fuente que se llama “pixelFont”.

Para hacer una prueba, vamos al archivo “**Play.js**” al método “create” e introducimos el siguiente texto:

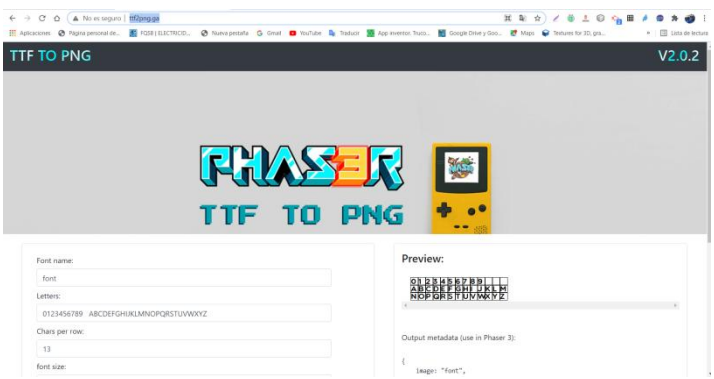
```
create() {
    this.add.bitmapText(100,100, 'pixelFont', 'PRUEBA');
}
```



En el navegador, debe aparecer en la “consola” que se ha iniciado la escena Play y aparecerá el texto de “PRUEBA”

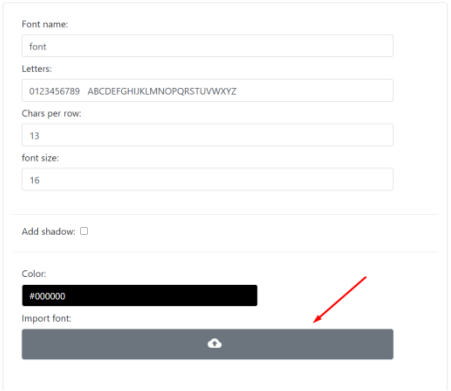
IMPORTANTE: Este texto debe aparecer en mayúsculas ya que lo hemos configurado para que escriba sólo palabras en mayúsculas al crear la fuente en http://ttf2png.ga/

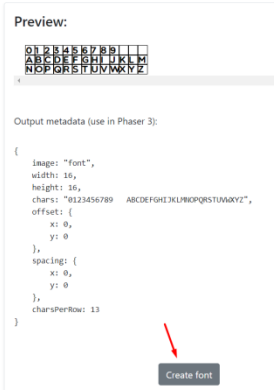
Para crear la fuente hemos utilizado la web descrita anteriormente:



Si quisieras configurar tu propia fuente de esta forma sólo tienes que introducir el nombre de la fuente que quieras en el campo “Font

name", los caracteres que quieras para la fuente (aquí podrías introducir también si quisieras minúsculas), en el campo "Letters". En el apartado "char per row" introduce los caracteres por línea (deja 13 por defecto; los verás en la vista previa)

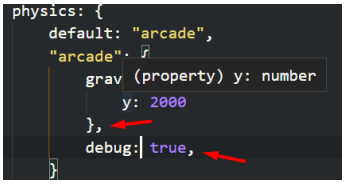




En el apartado "Import Font" arrastra el archivo "ttf" de la fuente que descargaste de https://fonts.google.com

Para finalizar puedes crear esta fuente pulsando en el botón "Create Font".

Una vez hecha esta prueba de texto, borro o comento esta línea con el texto de prueba y en el archivo "**main.js**" activo el "debug" para las físicas:



Nota: No olvides poner la coma después de la llave de la propiedad "gravity".

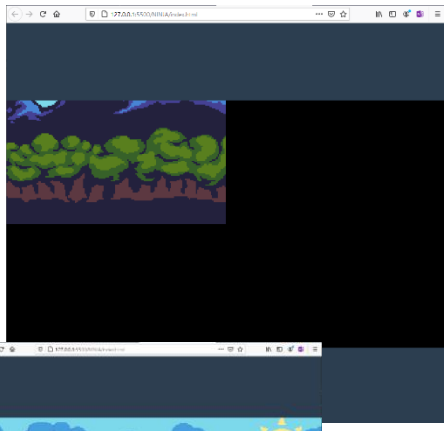
Y en los estilos del archivo "index.html" configuramos el fondo en negro:

```
body {
    background-color: #2c3e50;
}
```

En la **escena Play.js** se añade la imagen de fondo:

```
create() {
    this.add.image(0, 0, 'background');
}
```

Pero observa que queda mal situada:



La coordenada se sitúa por defecto en el centro de la imagen si cambiamos a la coordenada 0,0 se situará correctamente:

```
create() {
    this.add.image(0, 0, 'background')
        .setOrigin(0);
}
```

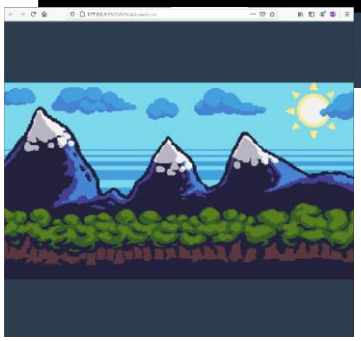


imagen ya se sitúa centrada correctamente➔

Vamos a agregar las paredes y el suelo con la imagen que hemos llamado “wall”:

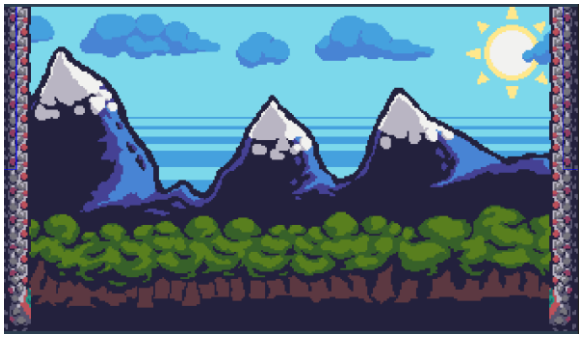
```
//Creamos las paredes y el suelo con un grupo
this.wall_floor = this.physics.add.staticGroup();

this.wall_floor.create(0, 0, 'wall')
.setOrigin(0);
```

Debemos situar su origen en (0,0)

Agregamos la pared de la derecha usando la anchura de la escala:

```
this.wall_floor.create(this.scale.width, 0 , 'wall')
    .setOrigin(1, 0)
    .setFlipX(true);
```

El origen se debe establecer en la esquina superior derecha (1,0) y la invertimos según el eje X para que se vea en “espejo” respecto de la pared izquierda.



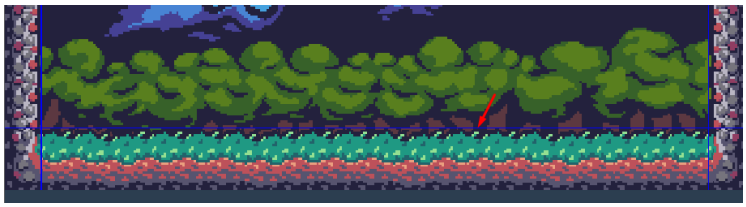
En la imagen de la izquierda se muestra la pantalla con las paredes y el suelo. Si te fijas en las colisiones (líneas azules) están mal situadas. Es porque en el grupo hay que refrescar estas colisiones al haber desplazado los orígenes:

```
//Refrescamos las colisiones
this.wall_floor.refresh();
```

Verás como ahora se verán las colisiones correctamente.

Añadimos el suelo al grupo:

```
//Suelo
this.wall_floor.create(0, this.scale.height, 'floor')
    .setOrigin(0,1);
```



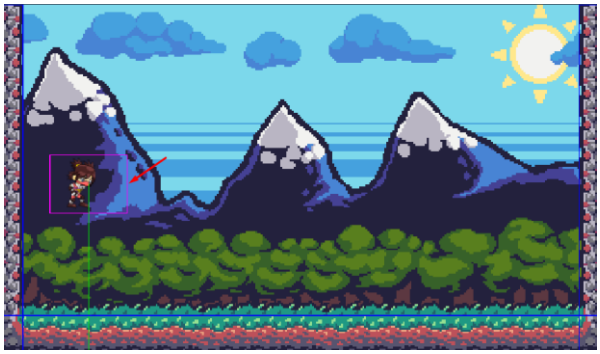
Vamos a bajar la colisión del suelo para que el jugador no parezca que flota por encima:

```
//Refrescamos las colisiones
this.wall_floor.refresh();
//Bajamos la colisión del suelo
this.wall_floor.getChildren()[2].setOffset(0, 15);
```

Cargamos el personaje:

```
//Player
this.ninja = this.physics.add.sprite(100, 100, 'ninja');
```

Y vemos que cae hacia abajo ➔

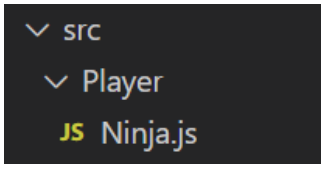


Para que choque con las paredes y el suelo debemos crear las colisiones:

```
//Colisiones con suelo y paredes
this.physics.add.collider([this.ninja], this.wall_floor);
```

Y vemos que cae al suelo, pero la colisión es demasiado grande con respecto al Sprite.

Para programar al personaje vamos a crear dentro de la **carpeta** "src" una llamada "**Player**" y en ella un archivo llamado "**Ninja.js**" ➔



En este archivo creo la clase "Ninja" que extiende de la clase "Sprite" que extiende a su vez de "GameObject":

```
class Ninja extends Phaser.GameObjects.Sprite{
    constructor(config){
        super(config.scene, config.x, config.y, 'ninja');
    }
}
export default Ninja;
```

En el constructor hay que modificar respecto de su clase superior (super) la escena, la coordenada "x" e "y" y su textura llamada 'ninja'. Exportamos la clase con "export default Ninja;"

Me voy a la escena "Play" y sustituyo la creación del personaje anterior por:

```
//personaje
this.ninja = new Ninja({
    scene: this,
    x: 100,
    y: 100
});
```

e importo en esta escena el archivo Ninja.js (No olvides la extensión):

```
import Ninja from "../Player/Ninja.js";
```

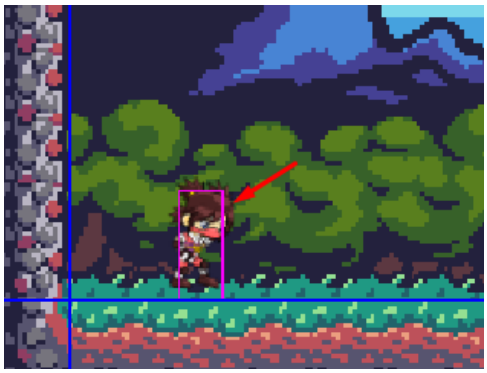
Nota: Se importa al inicio del archivo, antes de la clase

Pero para que se vea tengo que indicar lo siguiente en la clase "Ninja":

```
class Ninja extends Phaser.GameObjects.Sprite{
    constructor(config){
        super(config.scene, config.x, config.y, 'ninja');

        this.scene = config.scene;
        this.scene.add.existing(this);
        this.scene.physics.world.enable(this);
```

Que la escena del Sprite es la escena de la configuración del juego y que a esta escena existente le vas a añadir el Sprite "Ninja" y además con físicas para que colisione con el mundo.

Si ahora guardo veo que se agrega al personaje con sus colisiones.



Añadimos los siguientes ajustes a la clase "Ninja.js" (estos valores dependen del Sprite):

```
this.body.setSize(20, 50);
this.body.setOffset(32, 10);
```

←Para ajustar más las colisiones con el Sprite.

Vamos a crear una variable para controlar cuando está saltando el personaje:

```
//Salto
this.jumping = false;
```

Se trata de una variable booleana que va a tener un valor de "false" cuando no esté saltando y la vamos a invertir a "true" cuando lo haga.

Por último, para que al caer el personaje rebote un poco añadiremos:

```
//Rebote
this.body.setBounce(.2);
```

Animaciones

En el archivo "Ninja.js" del personaje cargo la animación en el constructor:

```
//Animación
this.anims.play('idle');
```

Crearemos una variable con la animación previa para saber qué animación se está reproduciendo previamente (si está saltando o en reposo) y una variable para hacer que sólo choque una vez contra el "enemigo".

```
//Animación previa
this.prevMov = 'idle';
//Variable para que sólo choque una vez con las estrellas
this.hitDelay = false;
```

Creamos las teclas Cursor para manejar al Player:

```
//Creamos las teclas Cursor
this.cursor = this.scene.input.keyboard.createCursorKeys();
```

Para empezar a moverlo de izquierda a derecha, creamos el método "update" dentro del archivo "Ninja.js" e incluímos los "if" necesarios:

```
update(){
    if(this.cursor.left.isDown){
        this.body.setVelocityX(-200);
    }else if(this.cursor.right.isDown){
        this.body.setVelocityX(200);
    }
}
```

Aún así no responde a las teclas porque no se está actualizando el update en el archivo “Play.js” para eso nos vamos a ese archivo y escribimos:

```
update() {
    this.ninja.update();
}
```

Ahora sí se mueve nuestro personaje.

Pero cuando pulsas una de las teclas (derecha o izquierda, al soltarla no deja de moverse) Esto lo vamos a solucionar añadiendo un “else” final que lo pare:

```
update(){
    if(this.cursor.left.isDown){
        this.body.setVelocityX(-200);
    }else if(this.cursor.right.isDown){
        this.body.setVelocityX(200);
    }else{
        this.body.setVelocityX(0);
    }
}
```

Cómo puedes ver en la imagen de la derecha, la condición es que no esté saltando y en ese caso paramos al personaje y disminuímos el tamaño y el “Offset” de la colisión. Incluímos en el “else” final el tamaño y el offset que teníamos en el “constructor”. ➔

Vamos a hacer que cuando se pulse la tecla de la flecha hacia abajo el personaje baje su colisión (como si se agachara) y al dejar de pulsarla vuelve a su posición normal.

```
update(){
    if(this.cursor.left.isDown){
        this.body.setVelocityX(-200);
    }else if(this.cursor.right.isDown){
        this.body.setVelocityX(200);
    }else if(this.cursor.down.isDown && !this.jumping){
        this.body.setVelocityX(0);
        this.body.setSize(20, 42);
        this.body.setOffset(32, 15);
    }
    else{
        this.body.setVelocityX(0);
        this.body.setSize(20, 50);
        this.body.setOffset(32, 10);
    }
}
```

Ahora vamos a hacer que salte. Para eso debemos asegurarnos que mientras salta aunque se vuelve a pulsar la flecha hacia arriba, hasta que no toque el suelo, no vuelva a saltar:

```
if(Phaser.Input.Keyboard.JustDown(this.cursor.up) && !this.jumping){
    this.jumping = true;
    this.body.setVelocityY(-800);
} else if(this.body.blocked.down){
    this.jumping = false;
}
```

La primera condición garantiza que sólo se pueda pulsar una vez y la segunda que sólo salte si la variable “jumping” es falsa (es decir, no está saltando). La condición del “else” cambia a “false” la variable “jumping” cuando el cuerpo del Player colisiona

(blocked) con su parte inferior (down). Pero el personaje no se anima ni se gira. Para eso vamos a programar lo siguiente:

```
if(this.cursor.left.isDown){
    this.body.setVelocityX(-200);
    this.flipX = true;
}else if(this.cursor.right.isDown){
    this.body.setVelocityX(200);
    this.flipX = false;
```

Para girar el personaje cuando camina hacia la derecha o hacia la izquierda modificamos la propiedad “flipX”.

Para las animaciones:

Habíamos creado la variable “prevMov” para controlar las animaciones. Si esta variable es distinta de ‘left” entra en la animación de “run” hacia la izquierda. Lo mismo para la derecha.

```
if(this.cursor.left.isDown){
    this.body.setVelocityX(-200);
    this.flipX = true;
    //Animación
    if(this.prevMov !== 'left' && !this.jumping){
        this.prevMov = 'left';
        this.anims.play('run');
    }
}else if(this.cursor.right.isDown){
    this.body.setVelocityX(200);
    this.flipX = false;
    //Animación
    if(this.prevMov !== 'right' && !this.jumping){
        this.prevMov = 'right';
        this.anims.play('run');
    }
```

Para la animación de agachar escribiremos cuando se pulsa la flecha hacia abajo:

```
if(this.prevMov !== 'down' && !this.jumping){
    this.prevMov = 'down';
    this.anims.play('slide');
}
```

Pero esta animación hay que volver a la animación “idle” cuando se deja de mover:

```
else{
    this.body.setVelocityX(0);
    this.body.setSize(20, 50);
    this.body.setOffset(32, 10);
    //Animación
    if(this.prevMov !== 'idle' && !this.jumping){
        this.prevMov = 'idle';
        this.anims.play('idle');
    }
```

En esta última animación no hay que poner la condición del salto ya que se trata de esta animación ➔

Y por último, me queda la animación de salto.

```
if(Phaser.Input.Keyboard.JustDown(this.cursor.up) && !this.jumping){
    this.jumping = true;
    this.body.setVelocityY(-800);
    //Animación
    if(this.prevMov !== 'jump'){
        this.prevMov = 'jump';
        this.anims.play('jump_attack');
    }
} else if(this.body.blocked.down){
    this.jumping = false;
}
```

Ahora, crearé una variable para las vidas al final del “constructor”:

```
//Vidas
this.life = 3;
```

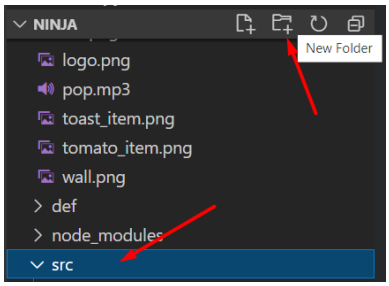
Y un método para las colisiones con las bombas.

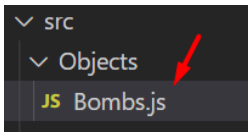
A partir de ahora vamos a incluir las bombas y las rebanadas de pan con manteca que debe ir recogiendo el Player y que le sumará puntos.

```
update(){···
}

bombCollision(){
    if(!this.hitDelay){

    }
}
```

Enemigos:

Los enemigos que van a ser bombas y las rebanadas de pan van a formar parte de grupos. Las bombas van a ir apareciendo cada vez que el Player recoge una rebanada. Para programar de forma ordenada y separada vamos a crear dentro de la carpeta "src" una carpeta llamada "Objects" y dentro un archivo llamado "bombs.js":





Y dentro vamos a extender de la clase de físicas de Arcade referida a los grupos.

```
class Bombs extends Phaser.Physics.Arcade.Group{

}
```

Para programar un grupo desde una clase extendida, se hace desde su constructor usando la configuración:

```
class Bombs extends Phaser.Physics.Arcade.Group{
    constructor(config){
        super(config.physicsworld, config.scene);
    }
}
```

Se pasa al constructor de la clase padre (Group de Arcade) con "super" los parámetros de esta clase (la física del mundo y la escena; physicsworld y scene)

Exportamos el archivo y lo importamos en el archivo "Play.js":

```
export default Bombs;
```

Antes del personaje en "create" creamos el grupo de las bombas y le pasamos los parámetros anteriores: "physicsworld" que será las físicas de la clase "Play" y su escena "this".

```
//Bombas
this.bombsGroup = new Bombs({
    physicsworld: this.physics.world,
    scene: this
});

//Player
//this.ninja = this.physics.add.sprite(100,
this.ninja = new Ninja({
```

```
class Bombs extends Phaser.Physics.Arcade.Group{
    constructor(config){
        super(config.physicsworld, config.scene);
        this.addBomb();
    }

    addBomb(){
        this.create(100, 100, 'bomb').setDepth(2);
    }
}
```

Para crear la primera bomba en el constructor de su clase vamos a llamar a un método llamado "addBomb" que vamos a crear nosotros. Fuera del "constructor" lo creamos y modificamos la propiedad "setDepth" a 2 para que no quede detrás la bomba de ningún otro objeto.

```
//Colisiones con el suelo y paredes
this.physics.add.collider([this.ninja, this.bombsGroup], this.wall_floor);
```

Para que colisione con las paredes y el suelo añado el grupo de las bombas al array anterior en la clase “Play”. Para que rebote, modifico en la clase “Bombs” al añadirla la propiedad setBounce y la pongo a “1”:

```
addBomb(){
    this.create(100, 100, 'bomb')
        .setDepth(2)
        .setBounce(1);
}
```

Vamos a agregar una colisión redondeada y que se desplace en función a hacia donde vaya girando. Para lo cual vamos a modificar varias propiedades del grupo:

“setCircle” a 18 para redondear la colisión, “setVelocityX” usamos un operador de aleatoriedad binaria (Phaser.Math.Between(0, 1)) y esto lo encerramos en unos paréntesis para usar un operador ternario para si es 1 que vaya a 100 y si es 0 que vaya a -100.

```
addBomb(){
    this.create(100, 100, 'bomb')
        .setDepth(2)
        .setBounce(1)
        .setCircle(18)
        .setVelocityX(
            (Phaser.Math.Between(0, 1)) ? 100 : -100
        );
}
```

Vamos a hacer que la gravedad de la bomba disminuya. Para eso modificamos la propiedad:

```
.setGravityY(-1800);
```

Vemos como rebota y cae con mayor lentitud.

Pero vamos a generar una bomba dentro de los límites de las paredes y el suelo:

```
addBomb(){
    this.create(
        Phaser.Math.Between(40, this.scene.scale.width -40)
        , -10, 'bomb')
        .setDepth(2)
        .setBounce(1)
        .setCircle(18)
        .setVelocityX(
            (Phaser.Math.Between(0, 1)) ? 100 : -100
        )
        .setGravityY(-1800);
}
```

En la coordenada x cambiamos el valor anterior (100) por la función aleatoria “Between” entre dos valores: 40 (ancho de la pared) y el ancho de la escena – 40. La coordenada y la situaremos en -10 para que caiga desde fuera del canvas.

Para que la bomba gire vamos a crear una función “update” e iremos “iterando” por todo el objeto, para que vaya repasando por todas las bombas, y vamos a comprobar si en “x2 va hacia la izquierda, que gire hacia ese lado y si va hacia la derecha que el giro se invierta.

```
update(){
    this.children.iterate( bomb => {
        if(bomb.body.velocity.x < 0){
            bomb.setAngularVelocity(-300);
        } else{
            bomb.setAngularVelocity(300);
        }
    });
}
```

En la clase “Play” tengo que actualizar el update de las bombas tal y como hice con el del “Player”:

```
update() {
    this.ninja.update();
    this.bombsGroup.update();
}
```

Colisión con las bombas:

```
//Colisión del Player con las bombas
this.physics.add.overlap(this.ninja, this.bombsGroup, () => {
    this.ninja.bombCollision();
});
```

Se crea la colisión pero con el método “overlap” y se usa un tercer parámetro que es una “Array function” que llama al método que creamos previamente en la clase “Ninja”.

```
bombCollision(){
    if(!this.hitDelay){
        this.hitDelay = true;
    }
}
```

En el método usé la variable “hitDelay” para que sólo se ejecute este método una sola vez (no cada vez que choca o se superpone la bomba). Para que esto ocurra, pongo esta variable a “true” cuando se ejecute este método. Vamos a agregar un evento temporal:

```
bombCollision(){
    if(!this.hitDelay){
        this.hitDelay = true;

        this.scene.time.addEvent({
            delay: 600,
            callback: () => {
                this.hitDelay = false;
            }
        });
    }
}
```

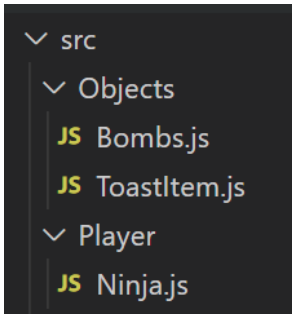
El retardo es de 600 milisegundos y al acabar el tiempo (callback) se ejecutará la función anónima que pone la variable “hitDelay” en valor “false”. Pero vamos a hacer que cuando choque reste una vida y emita dos eventos: uno para que remueva una vida (1) y otro para que si el número de vida es “0” ejecute un evento llamado ‘game_over’ (2). También vamos a hacer que cuando colisione cambie de color el personaje (3) y cuando acabe la colisión vuelva a su color (4).

```
bombCollision(){
    if(!this.hitDelay){
        this.hitDelay = true;

        this.life--;
        this.scene.registry.events.emit('remove_life');

        if(this.life === 0){
            this.scene.registry.events.emit('game_over');
        }

        this.setTint(0x1abc9c);

        this.scene.time.addEvent({
            delay: 600,
            callback: () => {
                this.hitDelay = false;
                this.clearTint();
            }
        });
    }
}
```

También vamos a reproducir un sonido cada vez que colisione (‘draw):

```
this.scene.sound.play('draw');
this.life--;
this.scene.registry.events.emit('remove_life');
```

Lo siguiente que haremos es agregar las tostadas para que cuando el personaje coja una aumente su puntuación y aparezcan más bombas. Para crear los “items” de rebanadas (archivo “toast_item.png”) hay que tener en cuenta el espacio interior del canvas quitando el de las paredes y el suelo y la altura que el personaje alcanza al saltar (en este caso es de 170) Para crear estos items, creamos dentro de la carpeta “Objects” un archivo llamado “ToastItem.js”.



Este grupo será un **grupo** con físicas pero **estático.**

```
class ToastItem extends Phaser.Physics.Arcade.StaticGroup{
    constructor(config){
        super(config.physicsworld, config.scene);
    }
}
```

Al constructor se le pasa el parámetro “config” y en a su clase superior (super) se le pasa los parámetros de las físicas del mundo y la escena.

```
class ToastItem extends Phaser.Physics.Arcade.StaticGroup{
    constructor(config){
        super(config.physicsworld, config.scene);
        this.addToastItem();
    }

    addToastItem(){
        this.create(
            Phaser.Math.Between(50, this.scene.scale.width -50),
            Phaser.Math.Between(150, this.scene.scale.height - 70),
            'toast_item'
        );
    }
}
```

Añadimos en el constructor el método para crear los items (addToastItem) y creo el método en el que la coordenada “x” es un número aleatorio entre dos valores: 50

(ancho de la pared y el ancho del canvas menos 50) y la coordenada “y” entre 150 (altura máxima que alcanza en el salto el ninja) y la altura del canvas menos 70 (altura del suelo) Creo un método para cuando el personaje recoje uno de los item:

```
destroyItem(){
    this.children.entries[0].destroy();
    this.addToastItem();
}
```

Con children.entries[0] accedo al primer item del grupo y con el método “destroy” lo destruyo. Al destruirse creo otro item llamando al método correspondiente. Exporto esta escena y la importo en el archivo “Play.js”:

```
export default ToastItem;
```

```
import ToastItem from '../Objects/ToastItem.js';
```

En el “create” después de las bombas añadimos los items:

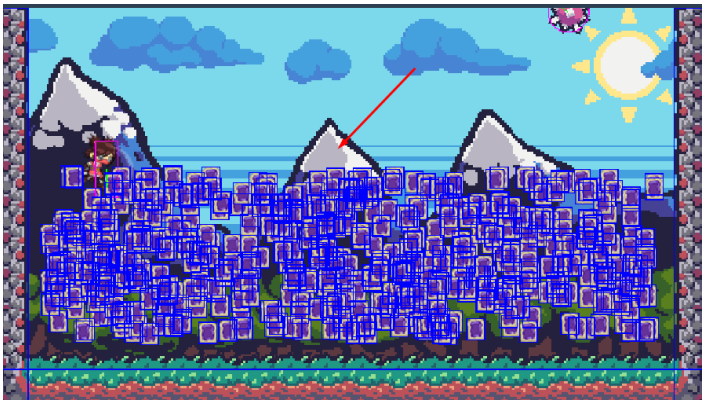
```
//Items
this.itemsGroup = new ToastItem({
    physicsworld: this.physics.world,
    scene: this
});
```

Se crea un nuevo objeto de “ToastItem” y pasamos los parámetros de físicas y escena tal y como hicimos con las bombas (copia y pega los parámetros de la bomba)

Para probar dónde se situan las bombas podemos hacer la prueba con una función que se reproduce a intervalos:

Se va a ejecutar la función anónima que llama al método para crear items cada 10 milisegundos. Si lo pruebas, verás que se crean los items dentro en el espacio que puede alcanzar el personaje al moverse y/o saltar.

```
constructor(config){
    super(config.physicsworld, config.scene);
    setInterval(() => {
        this.addToastItem();
    }, 10);
}
```



Volvemos a quitar la función a intervalos y dejamos la función “addToastItem”

```
constructor(config){
    super(config.physicsworld, config.scene);
    /* setInterval(() => {
        this.addToastItem();
    }, 10);  */
    this.addToastItem();
}
```

En la escena “Play” creamos las colisiones de los items con el personaje:

```
//Colisiones con los items
this.physics.add.overlap(this.ninja, this.itemsGroup, () => {
    this.sound.play('pop');
    this.registry.events.emit('update_points');
    this.itemsGroup.destroyItem();
    this.bombsGroup.addBomb();
});
```

Reproducimos un sonido, registramos la emisión de un evento que vamos a crear y que llamaremos “update_points”, destruimos el item y añadimos una bomba.

Marcador de puntos:

En la escena “Play.js” en el método “init” vamos a lanzar la escena “UI”:

```
init() {
    console.log('Se ha iniciado la escena Play');
    this.scene.launch('UI');
}
```

Esta escena “UI” se ha lanzado en paralelo y puedo pintar los objetos en esta escena.

En el “create” de la escena “UI” creareamos un grupo con los tres corazones que representarán las tres vidas del/la jugador/a.

```
create() {
    this.groupLife = this.add.group({
        key: 'life',
        repeat: 2,
        setXY: {
            x: 50,
            y:20,
            stepX: 25
        }
    });
}
```

En este caso no hace falta que sea un grupo con físicas. Comprobarás que no se ven los corazones porque en la escena “Main” vemos que la escena “Play” está por delante de “UI”:

```
scene: [
    Bootloader,
    UI,
    Play,
    Menu
]
```

Para pasarla delante escribimos en el método “init” de la escena “UI”:

```
init() {
    console.log('Se ha iniciado la escena UI');
    this.scene.moveUp();
}
```

Con el método “moveUp” pasamos la escena hacia delante. Ahora ya se ven los corazones.

Para añadir el marcador usamos la siguiente variable:

```
//Marcador con puntos
this.points = this.add.bitmapText(
    this.scale.width -40,
    20,
    'pixelFont',
    Phaser.Utils.String.Pad('0', 6, '0', 1)
).setOrigin(1, 0).setTint(0x000000);
}
```

Los dos primeros parámetros son las coordenadas a e y. El siguiente es el tipo de fuente, por último usamos la clase

String de Phaser con su método “Pad” que requiere los siguientes parámetros:

- El primer carácter con el que empieza
- Cuantos lugares queremos que se rellenen
- Que String hay que rellenar
- La dirección de rellenado (1 izquierda, 2 derecha, 3 ambas)

Con “setOrigin(1, 0) situamos el origen en la esquina superior derecha para que se sitúe correctamente y con “setTint(0x000000)” coloreamos los caracteres de negro para que contrasten más con el fondo del juego. Creo una variable actualizada a cero pero los puntos en el método “init”:

```
init() {
    console.log('Se ha iniciado la escena UI');
    this.scene.moveUp();
    this.actual_point = 0;
}
```

Vamos a recoger los eventos emitidos anteriormente. Primero por el “Player”. Seguimos en el “créate” de la escena “UI”:

```
//Eventos
this.registry.events.on('remove_life', () => {
    this.groupLife.getChildren()[this.groupLife.getChildren().length - 1].destroy();
});
```

Programamos un “escuchador” del evento ‘remove_life’ programado y ejecutamos la función anónima que recoge el último de los ítems del array de corazones (del grupo) restándole a la longitud de este array una unidad ya que el primer elemento es el índice cero. Este elemento lo destruimos con el método “destroy”.

Hacemos algo similar para el evento ‘game_over’:

```
//Evento game_over
this.registry.events.on('game_over', () => {
    this.registry.events.removeAllListeners();
    this.scene.start('Menu');
});
```

En este caso, removemos todas las escuchas de eventos y damos comienzo a la escena “Menu” creada en el archivo “js” del mismo nombre.

Para que sume puntos creamos otro escuchados de eventos para el evento ‘update_points’:

```
//Sumar puntos
this.registry.events.on('update_points', () => {
    this.actual_point += 10;
    this.points.setText(Phaser.Utils.String.Pad(this.actual_point, 6, '0', 1));
});
}
```

Y para que se pueda ver en el marcador, actualizo el texto, reutilizando el método “Pad” y cambiando el carácter “0” por los puntos (this.actual_point).

Dado que ya están programadas las colisiones, comentamos en el archivo “Main.js” el “debug” en la configuración de las físicas del juego:

```
physics: {
    default: "arcade",
    "arcade": {
        gravity: {
            y: 2000
        },
        //debug: true
    }
},
```

Escena “Menu”:

Vamos a ir al archivo “Bootloader.js” y en lugar de iniciar la escena “Play” al completarse la carga vamos a cargar la escena “Menu” y también vamos a reproducir el sonido ambiental del juego:

```
this.load.on('complete', () => {
    const fontData = this.cache.json.get('fontData');
    this.cache.bitmapFont.add('pixelFont', Phaser.GameObjects.RetroFont.Parse(this, fontData));

    this.scene.start('Menu');
});
```

Este sonido recibe un segundo parámetro en modo “json” del loop para que se reproduzca de forma indefinida.

```
this.load.on('complete', () => {

    //Sonido ambiental
    this.sound.play('bongo', {loop: true});
```

Como aún no tenemos configurado la escena “Menu” no se ve aún nada. Vamos a configurarla. Para empezar cargo las imágenes en el método “create”. Copio de la escena Play la carga de las paredes y el suelo:

```
create() {
    this.wall_floor.create(0,0, 'wall')
        .setOrigin(0);
    this.wall_floor.create(this.scale.width, 0, 'wall')
        .setOrigin(1,0)
        .setFlipX(true);
    //Suelo
    this.wall_floor.create(0, this.scale.height, 'floor')
        .setOrigin(0,1);
}
```

Cambio el “this.wall_floor.create” por “this.add.image”:

```
create() {
    this.add.image(0,0, 'wall')
        .setOrigin(0);
    this.add.image(this.scale.width, 0, 'wall')
        .setOrigin(1,0)
        .setFlipX(true);
    //Suelo
    this.add.image(0, this.scale.height, 'floor')
        .setOrigin(0,1);
}
```

Añadimos el fondo del juego:

```
create() {
    this: this
    this.add.image(0, 0, 'background').setOrigin(0);

    this.add.image(0,0, 'wall')
        .setOrigin(0);
```

Vamos a añadir el logo para que al clicar sobre él comience el juego:

```
this: this
this.logoMenu = this.add.image(
    this.scale.width/2,
    this.scale.height/2,
    'logo'
).setScale(2).setInteractive();
```

Lo ampliamos al doble de su tamaño y lo hacemos interactivo.

Vamos a hacer una salida del logo cuando se pulse sobre él:

```
//Inicio el tween de salida del menú
this.logoMenu.on(Phaser.Input.Events.POINTER_DOWN, () => {
    this.add.tween({
        targets: this.logoMenu,
        ease: 'Bounce.easeIn',
        y: -200,
        duration: 1000,
        onComplete: () => {
            this.scene.start('Play');
        }
    });
});
```

Es un evento que al pulsar sobre el logo ejecuta la función anónima que contiene el “tween”.

Cuando se completa el “tween”, se ejecuta la escena “Play”.

En el método “créate” de la escena “Menu” añadimos:

```
//Escribimos los puntos con texto
this.pointsText = this.add.bitmapText(
    this.scale.width/2,
    this.scale.height - 100,
    'pixelFont',
    'PUNTOS ' + this.points
).setDepth(2).setOrigin(.5);

//Inicio el tween de salida del menú
```

Este es el texto reservado para la puntuación. Como verás se empleó el ancho y la altura del canvas para situar las coordenadas, la fuente ‘pixelFont’ y lo pusimos delante del todo (setDepth(2)) y con el origen desplazado (setOrigin(.5)).

Vamos al método “init” de la escena “Menu” y iniciamos una variable para los puntos con valor inicial de “0” ➔

```
init() {
    //console.log('Se ha iniciado la escena Menu');
    this.points = 0;
}
```

Vamos a modificar el método “init” de la clase “Menu” para que reciba datos de otras escenas. Para eso, introducimos este parámetro y el siguiente código:

Con “Object.keys(data).lenght” comprobamos cuántos datos se enviaron y si es distinto de “0” es porque se envió alguno (hay que comprobar que se envió alguno para evitar errores)

```
init(data) {
    //console.log('Se ha iniciado la escena Menu');
    this.points = 0;
    //Comprobamos si se enviaron datos desde otras escenas
    if(Object.keys(data).length !== 0){
        this.points = data.points;
    }
}
```

En la escena “UI” en el evento ‘game_over’ añadimos el envío de puntos:

```
//Evento game_over
this.registry.events.on('game_over', () => {
    this.registry.events.removeAllListeners();
    this.scene.start('Menu', {points: this.actual_point});
});
```

En la escena ”Menu”, en el “create” añadimos para la mejor puntuación:

```
create() {

    const pointDB = localStorage.getItem('best_point');
    this.bestPoint = (pointDB !== null) ? pointDB : 0;
```

Debajo del código donde dibujamos el texto con los puntos, escribimos:

```
//Incluyo el texto para la mejor puntuación
this.bestPointsText = this.add.bitmapText(
    this.scale.width/2,
    this.scale.height - 80,
    'pixelFont',
    'MEJOR ' + this.bestPoint
).setDepth(2).setOrigin(.5);
```

Es un código similar pero cambiamos el texto por “MEJOR” y la variable a concatenar por “this.bestPoint” ➔

Tras el inicio del “tween” de salida del menú establezco la lógica para la mejor puntuación:

```
if(this.points > this.bestPoint){
    localStorage.setItem('best_point', this.points);
}
```

Añado el tween para las dos puntuaciones:

```
//Inicio el tween de salida del menú
this.logoMenu.on(Phaser.Input.Events.POINTER_DOWN, () =>
    this.add.tween({
        targets: this.logoMenu,
        ease: 'Bounce.easeIn',
        y: -200,
        duration: 1000,
        onComplete: () => {
            this.scene.start('Play');
        }
    });
    this.add.tween({
        targets: [this.pointsText, this.bestPointsText],
        ease: 'Bounce.easeIn',
        y: 400,
        duration: 1000,
    });
});
```

Con esto hemos completado nuestro video juego. Espero que te haya gustado.